Efficient nested locking in EOS

Laurent Daynes, Olivier Gruber

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Efficient nested locking in EOS

Laurent DAYNÈS
Olivier GRUBER

N° 1829
Janvier 1993

PROGRAMME 1

Architectures parallèles,
Bases de données,
Réseaux et Systèmes distribués

Efficient Nested Locking in Eos*

Laurent Daynès, Olivier Gruber
Projet Rodin
INRIA, Rocquencourt
78153 Le Chesnay Cedex, France

{daynes, gruber}@rodin.inria.fr

Abstract

Object-oriented languages, extended with nested actions, are an attractive basis for access to shared and persistent object stores in a distributed system. Unfortunately, existing implementations of nested actions are quite expensive. While there is a large body of work on durability and nested atomicity properties, very little exists on reducing the overhead of nested isolation. In this paper, we describe a new interpretation of the Moss locking rules, which enables a fast implementation of nested locking. Our measurements show that we outperform other nested locking implementations such as Argus or Camelot. Moreover, our performance is comparable to that of flat locking implementations such as Starburst. The key points of our approach are: greatly simplified conflict detection, an almost free upward and downward inheritance of locks (even in the presence of distributed action hierachies), and new locking support that is optimized for to cope with the high performance of single-level stores.

Key words : nested transaction, object locking, single-level store, distributed system, micro-kernel.

Verrouillage Emboîtés Efficace dans Eos.

Les langages orientés objets, étendus avec des actions emboîtées, constituent une approche attrayante pour contrôler l'accès à des espaces partagés et persistants d'objets dans un système distribué. Malheureusement, les réalisations existantes d'actions emboîtées sont très chères. Alors qu'il existe un nombre important de travaux sur les aspects durabilité et atomicité des actions emboîtées, très peu de travaux relatifs à la réduction des coûts de l'isolation emboîtée ont été publiés.

Dans ce papier, nous décrivons une nouvelle interprétation des règles de verrouillages de Moss qui nous permet une réalisation efficace. Nos mesures montrent que nous surpassons les autres réalisations de verrouillages emboîtés telles que celles d'Argus ou de Camelot. De plus, nos performances sont comparables à celles de réalisations de verrouillage en transactions plates telle que celle de Starburst. Les points clés de notre approche sont : une détection de conflit grandement simplifiée, de l'héritage ascendant et descendant de verrous pratiquement gratuit (même dans le cas de hiérarchies distribuées d'actions), et un nouveau support pour le verrouillage, adapté aux hautes performances des mémoires virtuelles persistantes.

Mots clés : transactions emboîtées, verrouillage objet, mémoire virtuelle persistante, système distribué, micro-noyaux.

*Submitted to SIGMOD '93 International Conference

1 Introduction

Object-oriented languages are an attractive basis for language-integrated and type-checked access to shared and persistent data and service in a distributed system. The object paradigm has been used in object-oriented database systems, persistent object stores, and distributed programming systems [6, 2, 15]. These systems are called Persistent Object Systems (POS) hereafter. POSs usually rely on an object store that provides a shared persistent object space over a distributed architecture. A distributed architecture is a set of computers (called nodes) interconnected through a communication network.

A commonly advocated design for supporting distribution is a fragmented store [16, 4, 8], i.e., the store is partitioned onto nodes and a function shipping paradigm is used to access remote objects. Furthermore, it is widely accepted that a single-level store is an efficient solution for supporting a persistent object store [5, 15]. Therefore, this suggests combining these two techniques, providing an efficient support for POSs. Unfortunately, POSs have to deal with chaos resulting from uncontrolled sharing of objects and failures. Nested transactions are often advocated for this purpose, but current implementations are very expensive [16, 4, 24, 25, 8].

In this paper, we propose a new implementation of nested locking. Our purpose is **not** to propose a new model. On the contrary, we only interpret the Moss model (with sibling parallelism) in a way that enables an efficient implementation. This work is part of the Eos system [12] which is a general-purpose programming environment for building multi-user softwares that need sharing, persistence and distribution capabilities. Eos is based on a microkernel such as Mach [26] or Chorus ones [23], which provides a well-suited context for optimizing the implementation of nested locking. Most of the materials presented herein do not necessitate microkernel support though.

The paper is structured as follows. Section 2 presents the general problem faced by POS designers when they consider nested actions. From that global view, we will focus on the single problem of nested locking. Section 3 describes a new interpretation of Moss's locking rules which dramatically reduces the compexity of conflict detection. Furthermore, it enables free inheritance of locks, both upward and downward. Section 4 describes our implementation of this new interpretation. Section 5 describes how we provide a new locking support suited for single-level stores where we pay particular attention to the critical issues of lock localization and latching. The rationale is, following [10], that Jim Gray's traditional locking support[11], first introduced for I/O bound systems and ever-used since, is too heavyweight for POSs. Section 6 presents performance measurements and Section 7 concludes.

2 The problem

Nested action models have been advocated for long due to their suitability for language-integrated access to shared and distributed object stores [4, 18, 8]. Nested actions provide the properties of Atomicity, Isolation, and Durability (AID)[1], but their support usually entails a high overhead.

In most implementations, atomicity and isolation properties of actions are obtained through locking and physical logging. Locks take time to request and space to manage. Physical logging induces some byte-copy overhead. Moreover, durability entails disk accesses when top-level actions commit to stabilize the effects of the committing hierarchies. To reduce this overhead is essential for POSs and constitutes the first challenge faced by POS designers.

Fortunately, the durability overhead can be neglected in the years to come. Durability was hurting performance drastically through the disk accesses that were necessary to stabilize commited data. With the availability of battery-backed-up memory or uninteruptible power supply units, durability could entail no major cost [9]. Therefore, durability will not be discussed any further in this paper.

To reduce the atomicity overhead suggests fine-grain logging. The smaller is the granularity, the smaller the byte-copy cost that is necessary to log before-images. Inasmuchas byte-copy cost is the major overhead of logging in the presence of battery-backed-up memory [9], fine-grain logging, such as object-grain logging, seems the key to performance. Recall that objects are language ones (class instances) and therefore small (dozens of bytes). Nested logging will not be further covered in this paper due to space limitation and to the large body of existing work in this area [22, 21, 1]. Indeed, the best schemes are fast and incur very minimal overhead when the logging granule is small and battery backed up memory is available.

Conversely to logging, locking suggests a coarse-grain in order to reduce its overhead. The larger is that grain the smaller the number of locks. Despite the fact that the granule of locking should be large, the usual hierachical locking based on a hierarchy of physical containers, such as pages and segments, is ill-suited to POSs. Indeed, locking at a coarser granule than the unit of sharing, which is the object in object-oriented languages, causes false conflicts, which in turn may degenerate in false deadlocks. A false deadlock example, when page-level locking is used, is illustrated in Figure 1. If one program (A) reads the white objects and another (B) writes the black ones, a deadlock occurs if both access their first object and then access the other one.

Furthermore, hierarchical locking is ill-suited for POSs because false conflicts make the correctness of parallel computations bound to the grouping of objects. In a POS, the grouping of objects should be periodically modified in order to reduce fragmentation and

[1] Consistency, the fourth property usually associated to transactions (ACID), is a responsability of users and not of the system providing actions.

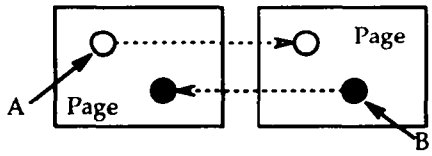


Figure 1: False Deadlock

to benefit from application locality. Hence, correctness of parallel programs is non deterministic, which might be a severe drawback. Therefore, we advocate that object grouping and locking issues should be kept orthogonal and hierarchical locking banished.

One may argue though that locking at an object granule results in too much overhead when objects are small. To solve this problem, several papers [14, 13] propose to use the semantics of complex objects. The basic idea is that fine grain locking enables a fine grain parallelism that is not always exploitable. Therefore, by structuring objects into complex objects and by associating a single lock to each complex object, the number of manipulated locks is significantly reduced. While grasping complex objects is a delicate data modeling issue, supporting them at an object store level is straightforward. It suffices to provide simple and lockable objects. Simple objects do not have a lock associated with them while lockable objects do. We will concentrate herein on the support for a fast implementation of lockable objects, leaving aside the modelling issue of complex objects.

To have such a fast implementation is essential for POSs because the overhead of locking is higher in POSs than in more specific systems such as relational database systems. On the one hand, POSs rely on single-level stores where objects are manipulated at memory speed. Therefore, any locking overhead erodes performance more significantly in POSs than in I/O bound systems. On the other hand, access patterns are unknown because algorithms are totally user-defined. Therefore, locks have to be checked for at almost every object access. In contrast, more specific systems usually optimize the number of times they request locks for the access patterns are perfectly known in advance like with the join, select and project operators in relational database systems.

Aside from the above first challenge met by POS designers, there is a second challenge which is nested specific. Indeed, nesting actions introduces specific locking overheads which are:

- More complex conflict detection.
- Repetitive upward and downward inheritance.

In the rest of this paper, we describe our threefold solution to these two challenges. First, we introduce a new interpretation of the Moss's locking rules which drastically reduces the complexity of conflict detection and further enables upward and downward inheritance of

locks at almost no cost. Second, we describe our nested action design in Eos which enables a simple and fast support of our interpretation of Moss locking rules. Third, we present how we efficiently support fine-grain locking in the context of single-level stores.

3 New Interpretation of Moss's Model

This section recalls briefly Moss's model, describes our earlier interpretation of it [7], and finally introduces a major extension to that interpretation which concerns the inheritance of locks.

3.1 Moss Model

The Moss model is based on the notion of actions which can be nested to arbitrarily depth. An action is a sequential thread of execution which executes at a single node. Actions are the basic unit of concurrent execution.

Actions are isolated and atomic. Isolated means serializable. Atomic means an action either completes successfully or has no effect at all, even in the presence of failures and deadlocks. Atomicity and isolation are traditionally ensured through locking and logging. Note that node-local actions provide node independence with respect to failures.

Actions can be nested in two ways. An action may call **synchronously** (respectively **asynchronously**) one or more subactions in parallel using specific constructs like co-begin [18]. A call is said to be synchronous (resp. asynchronous) when the caller waits (respectively does not wait) until the completion of all callees. Hierarchies of synchronously invoked actions are spheres of isolation and atomicity. In other words, action hierarchies introduce a recursive nesting of spheres of isolation and atomicity based on synchronous calls of actions. Asynchronous calls create new action hierarchies. In this context of action hierarchies, the locking rules are the following.

- An action can be granted a read lock if all write holders and write retainers of the lock are ancestors[2] of that action.
- An action can be granted a write lock if all holders and retainers of the lock (both read and write ones) are ancestors of that action.

3.2 Earlier Interpretation

Our earlier interpretation [7] of Moss's locking rules is based on tree and set theories. It includes both conflict detection and inheritance of locks and aims at permitting a fast implementation.

[2]The set of ancestors of an action includes that action.

In this interpretation, each action has a system-wide identity. Furthermore, each lock is a pair of sets of action identities, namely ReadSet (noted R_{owners}) and WriteSet (noted W_{owners}). The ReadSet (respectively WriteSet) of a lock holds the identities of owner[3] actions in read (resp. write) mode for that lock. Our rules for the detection of conflicts upon a given lock, for an action a_i, are given below. Note that these rules work in the presence of sibling parallelism but no parent/child parallelism.

- **Conflict detection on a read lock request.**

$$W_{owners} \subseteq Ancestors(a_i)$$

- **Conflict detection on write lock request (including the upgrade case).**

$$(W_{owners} \bigcup R_{owners}) \subseteq Ancestors(a_i)$$

This interpretation supports upward inheritance in a classical way but provides free support for downward inheritance. Following Moss's model, the support for upward inheritance is quite simple. A commiting action exchanges its own identity in all locks that it holds or retains with the identity of its parent action. Given that, downward inheritance is inherently supported by our conflict detection rules. In other words, downward inheritance causes no real downward flow of locks, but only a conflict detection based on the knowledge of the ancestor sets. A set-oriented conflict detection can be implemented efficiently so as to introduce negligeable overhead. Performance measurements will confirm that.

3.3 New Extended Interpretation

In our previous interpretation, both upward and downward inheritance is supported. Downward inheritance is efficiently supported but upward inheritance is still costly since it requires scanning all the locks of commiting actions. This section describes an extension to our previous interpretation that makes both upward and downward inheritance free. Having those mechanisms for free makes the commit of subactions also free. Therefore, we are able to support nested locking without any major overhead added to the non nested case[4]. Again, performance will confirm that.

In order to fully replace the lock flows going up and down the action hierarchy by flows of action identities, we need to define the following sets of action identities. Each action has a Upward Inherited Identity Set (UIIS), which is the union of its commited child identities and their UIIS.

$$UIIS(a_i) = \bigcup_j (a_j \cup UIIS(a_j))$$

[3] Owner actions of a lock are those that either hold or retain that lock.

[4] Recall that we do not consider logging, which in contrast to locking, incurs repetitive byte-copy costs in nested models.

where the a_j actions are the commited child actions of a_i.

Each action has a Downward Inherited Identity Set (DIIS), which is its parent identity plus the union of both the UIIS and DIIS of its parent action.

$$DIIS(a_i) = \bigcup_j (a_j \cup UIIS(a_j) \cup DIIS(a_j))$$

where a_j is the parent action of a_i.

Having the DIIS and UIIS sets of action identities, our conflict detection rules, for a given action a_i, become:

- **Conflict detection on a read lock request.**

$$W_{owners} \subseteq (DIIS(a_i) \cup a_i \cup UIIS(a_i))$$

- **Conflict detection on write lock request (including the upgrade case).**

$$(W_{owners} \bigcup R_{owners}) \subseteq (DIIS(a_i) \cup a_i \cup UIIS(a_i))$$

Let us illustrate in Figure 2 how this works on a simple inheritance case. The figure shows both inheritance mechanisms: in the upper half is our previous scheme while in the lower part is the new extented one. Let a_1 be an action that invokes two subactions a_2, a_3 successively. In a first step, the action a_2 is invoked and it requests read locks on objects x and y. Notice the action identifier (a_2) appears in the ReadSets of both x and y locks. In step two, the action a_2 commits.

- In the upper part, the action a_2 passes up its locks to its parent action by exchanging, in all the lock that it owns, its own identity by its parent identity. This takes place during the commit of action a_2.
- In the lower part, the action a_2 only adds up its identity to its parent UIIS. Notice that lock owner sets of locks are unchanged in this case.

In step 3, the parent action starts its second child action a_3.

- In the upper part, notice the ancestor set is equal to the pair composed of the parent and child action identity.
- In the lower part, notice the DIIS of the action a_3 contains the identities of both the parent action and the commited sibling a_2.

In both cases, if the action were to request locks on x and y, either in a read or write mode, they would be granted. This is easily verified by applying our conflict detection rules. However, there is a significant difference between the overheads of our two interpretations. A scan of all the lock owned by the action a_2 is necessary in our first interpretation while only a UIIS update is necessary in our second one. It is clear that our new interpretation permits almost free upward and downward inheritance.

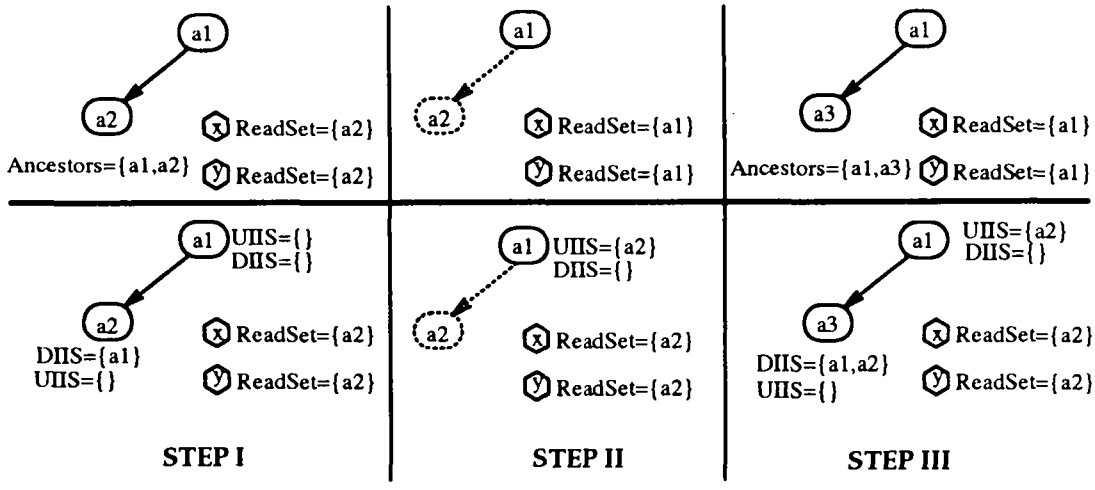


Figure 2: Upward and Downward Inheritance Example

4 Action Set Support

In this section, we focus on the support for action sets, which is a crucial issue in the context of our set-oriented interpretation of Moss's locking rules. We retained bitmaps where each bit identifies an action. The rationale is that bitmaps are well-suited for supporting our set-oriented manipulations of action sets for these manipulations only require regular logical operators such as $\vee$ or $\wedge$ operators.

If bitmaps seem an adequate choice from an operational point of view, their performance is highly related to their size. Thus, our challenge is to have the smallest bitmaps possible. Our solution to the size problem is detailed in the next two subsections. It relies on the fragmented nature of our object store and on a dynamic implementation of bitmaps.

4.1 Fragmentation

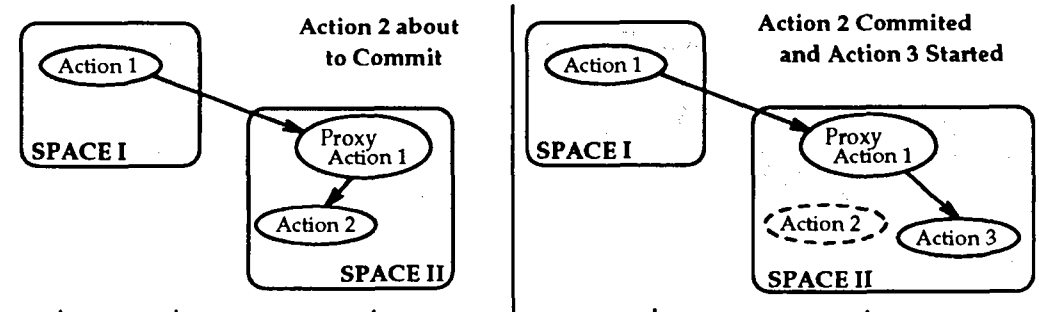
In our design, the object store is fragmented into spaces which are sets of (non contiguous) virtual pages. Each space is located at a single node, where its pages are stored on local disks. Recall there is no replication and that a function shipping paradigm is used instead. Having spaces, the idea is to make actions local to spaces and to name them locally. Local means that actions do not span multiple spaces and therefore a subaction is created each time a method invocation crosses a space boundary.

The names of actions running at a space are bit numbers. These bit numbers are space-wide, that is meaningful only in the action's space. This approach enables us to reduce bitmaps to praticable sizes. In the first place, the object space can be fragmented to node-sized spaces if there are sufficiently few actions per node. If too many actions are running per nodes, nothing precludes us from further fragmenting the object space by having multiple

spaces per node.

At first sight, it may sound strange in the presence of upward and downward inheritance that spaces only need to name local actions. Indeed, given an action having a remote subaction, i.e. running within another space, one could believe that locks would be upward inherited through space boundaries. However, recall that we do not have lock flows in our model but action identity flows instead. Moreover, having a fragmented approach without replication, it would be useless for information about lock owners to cross space boundaries since this information is needed where locked objects are. All this suggests a local mean for supporting upward inheritance of remote parent actions. For this purpose, action proxies are created which are action place-holders, upward inheriting *locally* the bits from local subactions on behalf of the remote parent action.

An example of upward inheritance with a proxy is depicted in Figure 3. On the left hand side of the figure, notice the proxy has no local bit (Local Bit Number, LBN) for it never request locks. On the right hand side of the figure, where the first child (Action 2) has commited and a second one (Action 3) has been invoked, notice the upward and downward inheritance of action identities. Upon the commit of the Action 2, the proxy upward inherited the bits of Action 2 (both LBN and UIIS). Upon the creation of the action 3, it downward inherited the bits in its DIIS from the local proxy. Finally, notice that both actions 1 and 3 have the same bit number (same LBN). This is perfectly correct since they are executing in different spaces.



	Action 1	Proxy Action 1	Action 2	Action 1	Proxy Action 1	Action 3
UIIS	000000	000000	010000	000000	010010	000000
DIIS	000000	000000	000000	000000	000000	010010
LBN	100000	000000	000010	100000	000000	100000

Figure 3: Upward Inheritance of Locks

Once an action has remote proxies, its commit is slightly complicated. As before, it has to update the UIIS of its parent action, however, this is a distributed process now since proxies are involved. Each commiting action multicasts a commit message to all of its proxies. These messages need only be gossip[5] messages, for no distributed synchronization

[5]A gossip message is a message sent asynchronously in background.

is necessary for commiting subactions. In particular, no two phase commit protocol is necessary. Of course, to commit a top level action and if failures are considered, proxies do incur a two-phase commit protocol.

Upon the reception of a commit message, a proxy tries passing up locally its UIIS to the parent action of its commiting action. If the parent action is local or already has a local proxy, the upward inheritance is done as usual. If the parent action is unknown locally, the proxy simply mutates into a proxy for the parent action. One can remark that a side effect of such mutations is that parent actions upward inherit proxies from their child actions. Thus, each subaction has to pass, when it commits, its proxy set to its parent action.

One final remark about our fragmentation-based scheme. A local naming of actions ensures small bitmap sizes if and only if bit numbers within spaces are recycled. Otherwise, bit numbers increase monotically making bitmap sizes to increase accordingly. Therefore, we have to recycle all the bits allocated to an action hierarchy when its top-level action commits. This is correct for when a top-level action commits, all the locks it retains or holds are freed, clearing its bits from lock owner sets.

4.2 Bitmap Specific Implementation

Albeit the size of bitmaps should be small, it may range from a couple of actions to a couple of hundreds per space. However, we believe that, most of the time, word-size bitmaps are enough, especially with the advent of 64bit machines such as the alpha or R6000. Having word-size bitmaps as the default representation enables very fast support for the logical operations needed by our conflict detection rules.

However, it is clear that we have to cope with overflows. A solution for overflow avoidance, often adopted in database servers, is to limit the number of actions concurrently executing within a space. The major advantage of this scheme is that it avoids thrashing, but in a nested model it might create hierarchical deadlocks between parent and child actions within a space: parents await their child actions which in turn wait for free bits while none are available. To solve this problem, we use timeouts. When an action waiting for a bit timeouts, we accept the overflow and allocate a bit number greater than 64. The rationale is that aborting is supposedly more costly than a temporary overflow of bitmaps. More about bitmap overflow will be said shortly.

One might say that our notion of temporary seems quite long. Indeed, the overflow state will last until overflow bits (greater than 64) can be reclaimed. In other words, the overflow state will last until the corresponding top-level actions commit. The overflow period can be reduced by introducing a dynamic reclaimation of action bits.

Once commited, actions mutate into a cleaning process (supported by the same thread) which reclaims the bits that the action owns. These bits, hereafter called owned bits of an action, are the action's own identity bit and the action's UIIS. To reclaim the owned bits of its action, a cleaning process scans all the locks owned by its action, exchanging these

bits with the bit of the parent action of its action. Once this is done, it updates the UIIS of concerned actions (parent and siblings) by removing these same bits, which can now be recycled. This scheme provides an early recycling of bits of commited subactions which should drastically limit or even suppress any latency to get free bits.

Let us detail now how bitmaps are overflowed by actions. Bitmaps can be overflowed only by actions whose bit number is greated than 64. We term these actions *overflow actions*. Overflow actions overflow lock owner sets when they are granted locks as depicted in Figure 4. A large bitmap is allocated. The original word-sized bitmap is copied in the first word of the newly allocated one, and that original bitmap becomes a pointer to the large one. Recall that overflowed bitmaps soon go back to a single word size because of the reclaimation of action bits.

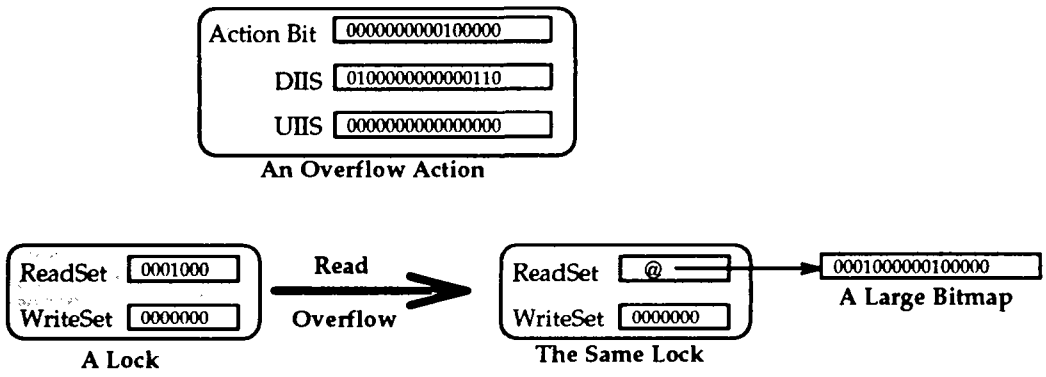


Figure 4: Owner Set Overflow

The critical point about an overflow mechanism is the dynamic allocation which usually requires taking a latch. Taking a latch is costly for this takes time and may create bottlenecks. We avoid this problem by providing each overflow action with a private allocator of large bitmaps. Therefore, there is no longer a need to take a latch.

5 Locking Support

In the previous section, we discussed an adequate implementation of locks based on bitmaps. However, the overhead of locking does not reduce to the overhead of conflict detection and lock inheritance. Locking also induces the following overheads: localization of the lock of an object; latching locks in order to avoid concurrent manipulation of locks by actions; book keeping of locks (lock set of actions). In the following subsections, we show how the context of a single-level store, implemented above a microkernel such as Mach [26] or Chorus [23] ones, enables to reduce these overheads to a minimum.

5.1 Lock Localization

Lock localization overhead should not spoil the advantages of single-level stores. Hence, Gray's traditional implementation of a lock manager must be rejected because hash table lookups to locate locks are too costly when objects are accessed at memory speed. A simple solution would be to include the lock within the object itself. However, this solution is out of consideration. Locks are always updated regardless of the access mode (read or write). Hence, if locks were stored along with objects, read-only actions would dirty all pages. Dirty pages reduce system throughput since they have to be written back on secondary storage.

Our solution is to access object locks via simple indirection. A virtual page is associated with a lock table. This table has one entry (lock) per object within the page. Furthermore, each object has a sequence number in its page. The lock of an object is accessed as follows. The address of the lock table is red from the page header. Then, the sequence number of the object is used as an index within the table. This approach has the following dangers:

- A test in the object locking path for the existence of lock table.
- To dirty pages by setting the lock table pointer in the page headers.
- Allocating a lock table for every virtual page that is not empty.
- Lock table fragmentation because of desequenced objects.

All these problems are easily solved above a microkernel where a single-level store is supported by a user-defined memory manager. In Mach terminology, a memory manager is a user-level process that backs up a region of virtual space. A memory manager fetches from disk the pages needed by the kernel to service page faults and also takes care of pages which are flushed out from the cache.

The memory manager abstraction is used in Eos as follows. Upon the very first fault on a page, the corresponding lock table is allocated and its address stored in the page header. Lock tables are deallocated when they are empty, i.e. all their locks are free, but only if their corresponding page is not present in the cache. Otherwise, the deallocation will be postponed until the page flush. In addition, the memory manager is also responsible for resequencing objects when necessary.

Our approach has the following advantages. Only virtual pages that are actually in the working set of currently running applications have a lock table. Furthermore, there is no need to test the existence of lock table, since they always seem in existence from the point of view of applications. Finally, pages are not dirtied by setting the pointers to lock tables because this is done before pages are even installed in the cache. Additionally, fragmentation is avoided because objects are re-sequenced when lock tables are deallocated.

5.2 Latching

Latching is necessary to control concurrent manipulations of locks by several actions. Again, speed is crucial since this mutual exclusion is around each lock manipulation which are themselves in the critical path of most object accesses. Moreover, the solution has to work in a multi-processor environment since small-scale multi-processors are likely to be the next generation of workstations.

Latching yields two questions. The first one is which granularity of mutual exclusion? The page sounds a natural granule in a small-scale multi-processors because the corresponding loss of parallelism should not be noticeable since the mutual exclusion is very short. The second question is how is mutual exclusion achieved?

Test and set is now a common assembler instruction in a wide variety of processors, it is fast and requires only one machine word of storage. For long, the test-and-set instruction has been known to have poor performance in multi-processors. This is no longer true though [20, 19]. However, test-and-set operations may significantly increase paging in persistent systems. Each test-and-set operation upon a memory word dirties the page it is in. Hence, if test-and-set operations are applied directly on store pages, paging is drastically increased. Similarly to locks, test-and-set words should be outside data pages. A natural location which still enables a fast access is in lock table headers.

5.3 Lock Book Keeping

Each action has to remember the set of locks it owns which is called its *locking set.* The rationales are two. One, a top-level action has, in order to commit, to free all its locks. Two, any action, in order to abort, has to undo all its effects both on objects and lock settings.

In Eos, each action maintains a stack of the addresses of lock tables where it owns locks. This stack of table addresses will be subsequently called the *locking set* of an action by assimilation. Locking sets are kept without double so that each lock table is processed only once when actions complete.

Doubles are easily avoided by maintaining in each lock table a set of actions which own at least one lock in that table. This set is called the *locker set.* Hence, when an action gets a lock, it checks if it appears in the locker set. If it does not, it adds the table address to its locking set and adds itself to the locker set of that table. The locking set of an action is depicted in Figure 5.

One can remark that having lock tables which are separated from data pages enables commiting top-level actions and aborting actions without accessing data pages. Hence, the cache management is free to replace virtual pages containing accessed objects based on any replacement policy without potentially causing costly faults during these "cleaning" processes.

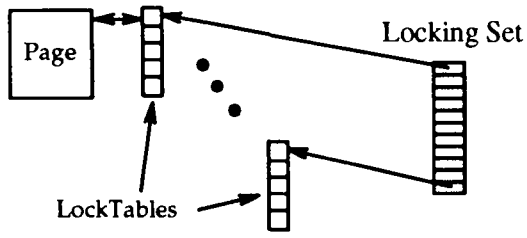


Figure 5: Locking Set

The rationale for implementing the lock sets of actions as stacks of lock table addresses instead of stacks of lock addresses is a matter of performance. Given that POS designs rely on an assumption of locality, we can assume that the number of locked objects per page is likely to be large. Therefore, remembering the lock table instead of each lock independently should be an advantage, even if it implies a complete scan of lock tables when lock sets are to be manipulated.

6 Performance Measurements

This section discusses the performances of our nested locking scheme. Performances in Eos are impacted on the one hand by our new support for locking and on the other hand by our new interpretation of Moss's locking rules. We will first compare ourselve with relational database systems, considering flat actions only, in order to evaluate the efficiency of our new locking support. The rationale is that relational systems are known for their efficient locking support. We will discuss next our support for nested locking, comparing ourselves with the two systems Argus [17] and Camelot [8].

All our measurements have been made on Sun 3/60 running the microkernel Mach 3.0 from Carnegie Mellon. Suns 3/60 are 3 MIPS machine, peak integer performance. Unfortunately, Suns are 32-bit machines and not 64-bit ones. To be fair to our design, we respected our assumption of word-sized bitmaps in order to have correct numbers of instructions in lock operations. The overflowed bitmaps are 128-bit wide.

As we were interested in the most common case where there was no lock contention, we measured lock cost by setting non-conflicting (in the nested sense) locks. All times given are an average of at least 10 runs over 100000 lockable objects unless stated otherwise. All times are given in microseconds. All times are memory-resident measures for we are interested only in measuring the locking overhead.

6.1 Flat Locking Support

Compared to the numerous papers on concurrency control methods, there is only one published work about changing the locking mechanism itself [10]. Jim Gray's design [11] seems to be the basis for most lock managers, including disk-oriented Starburst. In contrast, the memory-oriented Starburst proposes a new implementation of a lock manager to match the speed requirements of memory-resident database systems. This new approach attempts, as the Eos one, to reduce the following costs which are high in Gray's design.

- Lock latching.
- Locating locks.
- Dynamic allocation of locks upon lock requests.

Roughly speaking, the quality of both Starburst and Eos designs are equivalent with respect to these issues. Both have a single latch in the critical path of lock manipulations. In Starburst, a latch is taken at the table level, which protects both the data and lock manipulation. In Eos, a latch is taken per lock table. Both have a direct access to locks from objects (a single indirection in Eos). Finally, both systems have a preallocation scheme of locks, dramatically reducing the costs of lock dynamic allocation.

All this explains that performance of both systems are in the same order of magnitude. In the table below, we give the measures of the lock/unlock operations for the Disk-Oriented Starburst, the Memory-Oriented Starburst, and the Eos system. The corrected measures are based on an estimation of the difference of processing power between our benchmarking machine (Sun 3/60) and the Starburst one. Starburst benchmarking were conducted on a IBM Risc System 6000 Model 530. This is a 25 MIPS machine, peak integer performance. Following the authors of the Starburst paper [10], it is not uncommon to estimate one CISC instruction at 1.5 to 2 RISC intructions. Therefore, we applied a correction factor of 4.76 (equal to $25MIPS/(3MIPS * 1.75)$).

Operations	Disk-Oriented Starburst		Memory-Oriented Starburst		Eos (Sun3/60)	
	R6000	Corrected	R6000	Corrected	Word-Sized	Overflowed
Lock	25μ	119μ	9μ	42.8μ	28.3μ	47.8μ
Unlock	12μ	57.1μ	15μ	71.4μ	30.9μ	54.4μ

Having the measures of the three prototypes enables to evaluate the performance gain introduced by new locking supports. According to [10], the lock manager implementation of the disk-oriented Starburst is a reasonable implementation compared to those of R^* or GAMMA. The memory-oriented Starburst can reduce the locking cost by 35% as a result of using more efficient data structures[6] and reducing the overhead of latching.

[6] Notice that we do not consider the lock escalating or de-escalating schemes of Starburst for there is no such optimizations in a general-purpose POS.

Latching overhead is reduced in the MO Starburst through table-level latching philosophy while the DO Starburst latches independently the table, its indices, and the locking information. Notice the measures of the lock operation given in the table above (from [10]) include the cost of a latch/unlatch pair only for the DO Starburst. According to [10], a latch/unlatch on a R6000 is between 7.3μ and 18μ, depending on processor cache hit ratio. Therefore, reducing latching accounts for roughly 12μ in the lock operation improvement, which is about the three quarter of it.

Compared to the MO Starburst, Eos seems to be able to further reduce locking by as much as 50%. Of course, since Starburst provides a flat model of transactions, this comparison concerns the no-inheritance case only. When bitmaps are word-sized, Eos performance are twice those of the MO Starburst, that is up to 64 actions per space. Recall there can be many spaces per node. Albeit Sun3/60 are 32-bit machines, the above statement is fair for the number of instructions that are necessary to manipulate word-sized bitmaps is invariant whatever is the machine word size. In the overflowed case, that is up to 256 actions (4 word bitmaps) on 64-bit machines, Eos has equivalent performance to that of MO Starburst.

Albeit the measures for the MO Starburst do not include any latching overhead while Eos measures do, the comparison is fair. On the one hand, latching/unlatching in Eos only accounts for 2.2μ since Sun3/60 have a test&set (TAS) instruction. On the other hand, according to Brian N. Bershad in [3], having a TAS instruction is no performance advantage above Mach 3.0 because restartable atomic critical sections outperform TAS-based ones on most uniprocessor hardware, including RISC architectures such as the R6000.

There seems to be only one major difference between the Eos and memory-oriented Starburst designs which can justify the performance improvement. This is the implementation of locks themselves: bitmap-based for Eos, and traditional linked list of lock control blocks for Starburst [10], suggesting that bitmaps are a more efficient solution.

6.2 Nested Locking Support

In order to evaluate the Eos support for nested locking, it seems natural to compare it with Argus and Camelot, two systems which also provide nested actions. Moreover, the three systems have more or less the same distributed design, that is a function shipping paradigm over a fragmented store.

Our measures, given in the table below, show that Eos is significantly faster than Argus and Camelot. The table gives corrected measures based on the following processing powers: Sun3/60 (Eos) 3 MIPS; MicroVax II (Argus) 0.9 MIPS; IBM PC RT 125 (Camelot) 4.5 MIPS. There are five basic measures in a nested locking scheme: acquiring, re-checking (requesting a lock that is already held), inheriting (acquiring a conflicting lock after successful check of downward inheritance), upward inheriting, and releasing a lock. Because bitmaps might be word-sized or overflowed and also because we have two conflict detection rules which do not incur the same overhead, the performance of the lock/unlock operation in Eos

varies between two precise bounds.

	Eos Min Time	Eos Max Time	Argus[a]	Camelot[b]
Acquiring	28.3μ	47.8μ	60μ	315μ
Re-checking	27.3μ	34.4μ	33μ	315μ
Downward Inheriting	39.1μ	73.2μ	-	495μ
Upward Inheriting	free op.	free op.	87μ	-
Release	30.9μ	54.4μ	63μ	270μ

[a]The correction facter is 3.33.

[b]The correction factor is 0.66.

There are several points to notice about these measures. The lock/unlock operations, without inheritance, in Argus are surprisingly fast compared to lock managers of relational database systems. In fact, Argus performance is in between the performance of the memory-oriented and the disk-oriented starburst prototypes. We can guess that the reason lies in the scheduling mechanism of Argus. Argus uses a non preemptive scheduler which is totally integrated with its thread package [17]. Therefore, threads are preempted at well-known points only, avoiding latching in lock manipulations. This seems to be confirm because Argus outperforms the disk-oriented Starburst by approximately the cost of a latch/unlatch. Recall a latch/unlatch on a R6000 is about 12μ (time uncorrected), which is roughly equivalent to 60μ once corrected for Sun3/60. However, Argus remains slower than the main-memory Starburst because it seems not to use direct pointers to locate object locks. As opposed to Argus, Camelot performance are very poor. The only reason we can think of is that they pay the extreme generality of their design, including the adoption of a library philosophy.

Aside from the above points concerning the no-inheritance locking operations, the performance of the lock operations in the presence of inheritance are interesting to discuss. However, the comparison between the three systems is delicate for the lock inheritance mechanisms that they provide differ greatly.

In Argus, the inheritance mechanisms differ depending on whether both the parent and child actions are local or remote. When they are both local, locks are upward inherited as in the traditional Moss model. The cost of 87μ given in the table above correspond to that case. When the parent action is remote, locks are neither upward inherited through the network nor by a local proxy. In other words, commited actions remain the holders of their locks. Therefore, when a conflict is detected, network communications are necessary to determine the true state of affairs, that is if the requester can downward inherit the lock or not. The cost is unknown and seems furthermore highly variable given the Argus's distributed conflict detection protocol.

In Camelot, locks are not upward inherited at commit time but on the contrary a lazy scheme is used. When a conflict occurs on a lock, the requester requests information about the outcomes of actions in the paths between current holders and the least common ancestors between these holders and itself. Each node upholds some local caching about action

outcomes so that network communications can be mostly avoided during this process [8]. If the conflict detection can be solved locally, the cost is 495μ (corrected time) given in the table above. Otherwise, network communications yield a cost of 4970μ (uncorrected).

In Eos, both the upward and downward inheritance of locks are eagerly supported, but incurring very little cost. Recall that upward inheritance is supported through an upward flow of action identities, which incurs a totally insignificant cost. Downward inheritance makes the conflict detection slightly more costly, but *no network communication ever occurs* in the locking path of objects.

Additionally, it is interesting to compare the Eos locking costs (involving downward inheritance) with the flat locking costs in both Starburst prototypes. One can remark that our best measure equals the measure of the memory-oriented Starburst, while our worse measure matches the measure of the disk-oriented Starburst, latching cost removed in order to be fair. The necessary measures for this comparison are recalled below.

Measured System	lock	
Corrected MO Starburst Measure	**42.8μ**	
Eos (inheritance cases)	Min Times	Max Times
	39.1μ	**73.2μ**
Corrected DO Starburst Measure without Latching	**59μ**	

7 Conclusion

In this paper, we discussed the implementation of nested actions in the context of persistent object systems. We described a new interpretation of the Moss locking rules that enables an efficient implementation. The key points of this interpretation are: a conflict detection that relies on simple set operations, and an inheritance of locks, both upward and downward, that relies on simple flows of action identities along the action hierarchy.

Our measures show that we are faster than other nested locking schemes we are aware of. In particular, notice that our conflict detection scheme never involves network communications. Moreover, our nested locking implementation outperforms those inspired on Gray's design which seems to be the basis for most lock managers. Furthermore, our performance is even comparable to that of the two lock managers of Starburst. Eos seems to be faster in the no-inheritance locking cases than the optimized lock manager for memory-resident databases [10], and to have analogous performance to the traditional lock manager.

Contributions of this work are not restricted though to the specific context of POSs, but apply to most lock manager designs. Future works clearly encompass optimizing nested atomicity and durability supports in order to cut the complete overhead of nested actions to a praticable level for POSs.

Acknowledgements We are especially grateful to Laurent Amsaleg for his help and support. Finally, we would like to acknowledge the continous encouragements of Patrick Valduriez in regards of Eos becoming.

References

[1] O. B., L. B., and S. R. Reliable Object Storage to Support Atomic Actions. In *10th Symposium Operating Systems Principles*. ACM, 1985.

[2] F. Bancilhon, C. Delobel, and P. Kannellakis. *Building an object-oriented database : the O2 story*. Morgan Kaufmann, 1991.

[3] J. R. E. Brian B. Bershad, David D. Redell. Fast Mutual Exclusion for Uniprocessors. In *Fifth Internationnal Conference on Architectural Support for Programming Language and Operating System*, pages 223–233, Boston, Massachussetts, October 1992.

[4] J. D. N. Calton Pu. Design and Implementation of Nested Transaction in Eden. In *Symposium on Reliability in Distributed Software and Database System*, pages 126–136. IEEE, March 1987.

[5] G. Copeland, M. Franklin, and G. Weikum. Uniform Object Management. In *Proc. of the Int. Conf. on Extended Database Technology*, Venice, Italy, 1990.

[6] O. corporation. Ontos. Product Description, March 1990.

[7] L. Daynès and O. Gruber. Nested Action in Eos. In *Fifth International Workshop on Persistant Object Systems*, pages 115–138, September 1992.

[8] J. Eppinger, L. Mummert, and A. Spector. *Camelot and Avalon*. Morgan Kaufmann, 1991.

[9] R. K. a. M. S. G. Copeland, T. Keller. The Case for Safe RAM. In *15th International Conference on Very Large Databases*, pages 327–335, 1989.

[10] V. Gottemukkala and T. J. Lehman. Locking and Latching in a Memory-Resident Database System. In *18th International Conference on Very Large Databases*, pages 533–544, 1992.

[11] J. Gray. Notes on Database Operating Systems. In *Operating Systems, An Advanced Course*, volume 60. Springer-Verlag, New York, 1978.

[12] O. Gruber, L. Amsaleg, L. Daynès, and P. Valduriez. Eos, An Environment for Object-Based Systems. In *Proc. of the 25th Hawaii International Conference on System Sciences*, volume 1, pages 757–768, January 1992.

[13] U. Herrmann, P. Dadam, K. Kuspert, E. A. Roman, and G. Schlageter. A Lock Technique for Disjoint and Non-Disjoint Complex Objects. In Springer-Verlag, editor, *Second International Conference on Extending Database Technology*, pages 219–237, 1990.

[14] W. K. Jorge. F Garza. Transaction Management in an Object-Oriented Database System. In *ACM SIGMOD, Conf. on Management of Data*, pages 37–45, 1988.

[15] C. Lamb, G. Landis, J. Orenstein, and D. Weinred. The Object Store Database system. *In Communications of the ACM*, 34(10):51, October 1991.

[16] B. Liskov, D. Curtis, P. Johnson, and R. Scheifler. Implementation of Argus. In *ACM Transactions on Programming Languages and Systems*, Cambridge, Ma, 1987.

[17] B. Liskov, D. Curtis, P. johnson, and R. Scheifler. Implementation of Argus. In *11th Symposium Operating Systems Principles*, pages 111–122, Austin,TX, November 1987. ACM SIGOPS.

[18] B. Liskov and R. Scheifler. Guardians and Actions : Linguistic support for robust, distributed programs. In *ACM Transactions on Programming Languages and Systems*. ACM, July 1983.

[19] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, February 1991.

[20] J. M. Mellor-Crummey and M. L. Scott. Synchronization without contention. In *ASPLOS, International Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 269–278, Santa Clara, CA (USA), April 1991.

[21] J. E. B. Moss. Log-Based Recovery for Nested Transaction. In *13th International Conference on Very Large Databases*, pages 427–432, Brighton, 1987.

[22] K. Rothermel and C. Mohan. ARIES/NT : A Recovery Method Based on Write-Ahead Logging for Nested Transactions. In *15th International Conference on Very Large Databases*, pages 337–346, Amsterdam, the Nederlands, August 1989.

[23] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Overview of the Chorus distributed operating system. In *Workshop on Micro-Kernels and Other Kernel Architectures*, pages 39–70, Seattle WA (USA), April 1992. Usenix.

[24] S. Shrivastava, G. Dixon, and G. Parrington. An overview of the Arjuna Distributed Programming System. *IEEE Software*, January 1991.

[25] V. Technology. *Versant in Brief*. Prentice-Hall, 1991.

[26] L. Walmer and M. Thompson. MACH documents. Technical report, Department of Computer Science, Carnegie-Mellon University, Pittsburg, January 1988.

Unité de Recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)

Unité de Recherche INRIA Lorraine Technopôle de Nancy-Brabois - Campus Scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 VILLERS LES NANCY Cedex (France)
Unité de Recherche INRIA Rennes IRISA, Campus Universitaire de Beaulieu 35042 RENNES Cedex (France)
Unité de Recherche INRIA Rhône-Alpes 46, avenue Félix Viallet - 38031 GRENOBLE Cedex (France)
Unité de Recherche INRIA Sophia Antipolis 2004, route des Lucioles - B.P. 93 - 06902 SOPHIA ANTIPOLIS Cedex (France)

EDITEUR
INRIA - Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)

ISSN 0249 - 6399